appendix b

From LC-3 to x86

As you know, the ISA of the LC-3 explicitly specifies the interface between what the LC-3 machine language programmer or LC-3 compilers produce and what a microarchitecture of the LC-3 can accept and process. Among those things specified are the address space and addressability of memory, the number and size of the registers, the format of the instructions, the opcodes, the data types that are the encodings used to represent information, and the addressing modes that are available for determining the location of an operand.

The ISA of the microprocessor in your PC also specifies an interface between the compilers and the microarchitecture. However, in the case of the PC, the ISA is not the LC-3. Rather it is the x86. Intel introduced the first member of this ISA in 1979. It was called the 8086, and the "normal" size of the addresses and data elements it processed was 16 bits. The typical size of addresses and data today is 32 bits. From the 8086 to the present time, Intel has continued implementations of this ISA, the 80286 (in 1982), 386 (in 1985), 486 (in 1989), Pentium (in 1992), Pentium Pro (in 1995), Pentium II (in 1997), Pentium III (in 1999), and Pentium IV (in 2001).

The ISA of the x86 is much more complicated than that of the LC-3. There are more opcodes, more data types, more addressing modes, a more complicated memory structure, and a more complicated encoding of instructions into 0s and 1s. However, fundamentally, they have the same basic ingredients.

You have spent a good deal of time understanding computing within the context of the LC-3. Some may feel that it would be good to learn about a *real* ISA. One way to do that would be to have some company such as Intel mass-produce LC-3s, some other company like Dell use them in their PCs, and a third company such as Microsoft compile Windows NT into the ISA of the LC-3. An easier way to introduce you to a *real* ISA is by way of this appendix.

We present here elements of the x86, a very complicated ISA. We do so in spite of its complexity, because it is the most pervasive of all ISAs available in the marketplace.

We make no attempt to provide a complete specification of the x86 ISA. That would require a whole book by itself, and to appreciate it, a deeper understanding of operating systems, compilers, and computer systems than we think is reasonable at this point in your education. If one wants a complete treatment, we recommend *Intel Architecture Software Developer's Manual*, volumes 1, 2, and 3, published by Intel Corporation, 1997. In this appendix, we restrict ourselves to some of the characteristics that are relevant to application programs. Our intent is to give you a sense of the richness of the x86 ISA. We introduce

these characteristics within the context of the LC-3 ISA, an ISA with which you are familiar.

B.1 LC-3 Features and Corresponding x86 Features

B.1.1 Instruction Set

An instruction set is made up of instructions, each of which has an opcode and zero or more operands. The number of operands depends on how many are needed by the corresponding opcode. Each operand is a data element and is encoded according to its data type. The location of an operand is determined by evaluating its addressing mode.

The LC-3 instruction set contains one data type, 15 opcodes, and three addressing modes: PC-relative (LD, ST), indirect (LDI, STI), and register-plus-offset (LDR, STR). The x86 instruction set has more than a dozen data types, over a hundred opcodes, and more than two dozen addressing modes (depending on how you count).

Data Types

Recall that a data type is a representation of information such that the ISA provides opcodes that operate on information that is encoded in that representation.

The LC-3 supports only one data type, 16-bit 2's-complement integers. This is not enough for efficient processing in the real world. Scientific applications need numbers that are represented by the floating point data type. Multimedia applications require information that is represented by a different data type. Commercial applications written years ago, but still active today, require an additional data type, referred to as *packed decimal*. Some applications require a greater range of values and a greater precision of each value than other applications.

As a result of all these requirements, the x86 is designed with instructions that operate on (for example) 8-bit integers, 16-bit integers, and 32-bit integers, 32-bit floating point numbers and 64-bit floating point numbers, 64-bit multimedia values and 128-bit multimedia values. Figure B.1 shows some of the data types present in the x86 ISA.

Opcodes

The LC-3 comprises 15 opcodes; the x86 instruction set comprises more than 200 opcodes. Recall that the three basic instruction types are operates, data movement, and control. Operates process information, data movement opcodes move information from one place to another (including input and output), and control opcodes change the flow of the instruction stream.

In addition, we should add a fourth category to handle functions that must be performed in the real world because a user program runs in the context of an operating system that is controlling a computer system, rather than in isolation. These instructions deal with computer security, system management, hardware performance monitoring, and various other issues that are beyond what the typical application program pays attention to. We will ignore those instructions in this

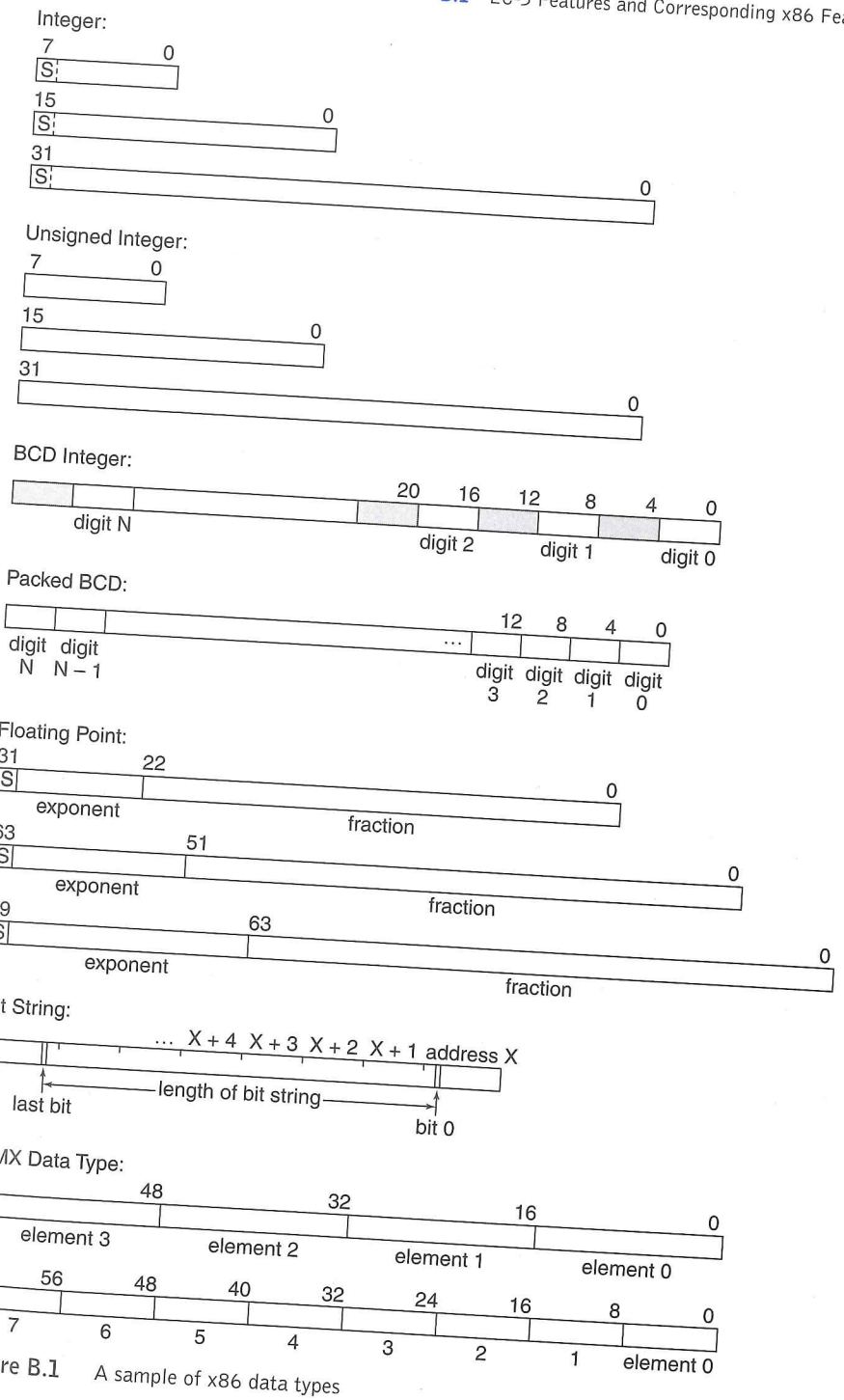

Figure B.1 A sample of x86 data types

appendix, but please note that they do exist, and you will see them as your studies progress.

Here we will concentrate on the three basic instruction types: operates, data movement, and control.

Operates The LC-3 has three operate instructions: ADD, AND, and NOT. The ADD opcode is the only LC-3 opcode that performs arithmetic. If one wants to subtract, one obtains the negative of an operand and then adds. If one wants to multiply, one can write a program with a loop to ADD a number some specified number of times. However, this is too time-consuming for a real microprocessor. So the x86 has separate SUB and MUL, as well as DIV, INC (increment), DEC (decrement), and ADC (add with carry), to name a few.

A useful feature of an ISA is to extend the size of the integers on which it can operate. To do this one writes a program to operate on such *long* integers. The ADC opcode, which adds two operands plus the carry from the previous add, is a very useful opcode for extending the size of integers.

In addition, the x86 has, for each data type, its own set of opcodes to operate on that data type. For example, multimedia instructions (collectively called the MMX instructions) often require *saturating arithmetic*, which is very different from the arithmetic we are used to. PADDS is an opcode that adds two operands with saturating arithmetic.

Saturating arithmetic can be explained as follows: Suppose we represent the degree of grayness of an element in a figure with a digit from 0 to 9, where 0 is white and 9 is black. Suppose we want to add some darkness to an existing value of grayness of that figure. An element could start out with a grayness value of 7, and we might wish to add a 5 worth of darkness to it. In normal arithmetic, $7 + 5$ is 2 (with a carry), which is lighter than either 7 or 5. Something is wrong! With saturating arithmetic, when we reach 9, we stay there—we do not generate a carry. So, for example, $7 + 5 = 9$ and $9 + n = 9$. Saturating arithmetic is a different kind of arithmetic, and the x86 has opcodes (MMX instructions) that perform this type of arithmetic.

Scientific applications require opcodes that operate on values represented in the floating point data type. FADD, FMUL, FSIN, FSQRT are examples of floating point opcodes in the x86 ISA.

The AND and NOT opcodes are the only LC-3 opcodes that perform logical functions. One can construct any logical expression using these two opcodes. However, as is the case with arithmetic, this also is too time-consuming. The x86 has in addition separate OR, XOR, AND-NOT, and separate logical operators for different data types.

Furthermore, the x86 has a number of other operate instructions that set and clear registers, convert a value from one data type to another, shift or rotate the bits of a data element, and so on.

Table B.1 lists some of the operate opcodes in the x86 instruction set.

Data Movement The LC-3 has seven data movement opcodes: LD, LDI, ST, STI, LDR, STR, and LEA. Except for LEA, which loads an address into a register,

Table B.1 Operate Instructions, x86 ISA

Instruction	Explanation
ADC x, y	x, y, and the carry retained from the last relevant operation (in CF) are added and the result stored in x.
MUL x	The value in EAX is multiplied by x, and the result is stored in the 64-bit register formed by EDX, EAX.
SAR x	x is arithmetic right is shifted n bits, and the result is stored in x. The value of n can be 1, an immediate operand, or the count in the CL register.
XOR x, y	A bit-wise exclusive-OR is performed on x, y and the result is stored in x.
DAA	After adding two packed decimal numbers, AL contains two BCD values, which may be incorrect due to propagation of the carry bit after 15, rather than after 9. DAA corrects the two BCD digits in AL.
FSIN	The top of the stack (call it x) is popped. The sin(x) is computed and pushed onto the stack.
FADD	The top two elements on the stack are popped, added, and their result pushed onto the stack.
PANDN x, y	A bit-wise AND-NOT operation is performed on MMX values x, y, and the result is stored in x.
PADDS x, y	Saturating addition is performed on packed MMX values x, y, and the result is stored in x.

they copy information between memory (and memory-mapped device registers) and the eight general purpose registers, R0 to R7.

The x86 has, in addition to these, many other data movement opcodes. XCHG can swap the contents of two locations. PUSHA pushes all eight general purpose registers onto the stack. IN and OUT move data between input and output ports and the processor. CMOVcc copies a value from one location to another only if a previously computed condition is true.

Table B.2 lists some of the data movement opcodes in the x86 instruction set.

Table B.2 Data Movement Instructions, x86 ISA

Instruction	Explanation
MOV x, y	The value stored in y is copied into x.
XCHG x, y	The values stored in x and y are swapped.
PUSHA	All the registers are pushed onto the top of the stack.
MOVS	The element in the DS segment pointed to by ESI is copied into the location in the ES segment pointed to by EDI. After the copy has been performed, ESI and EDI are both incremented.
REP MOVS	Perform the MOVS. Then decrement ECX. Repeat this instruction until ECX = 0. (This allows a string to be copied in a single instruction, after initializing ECX.)
LODS	The element in the DS segment pointed to by ESI is loaded into EAX, and ESI is incremented or decremented, according to the value of the DF flag.
INS	Data from the I/O port specified by the DX register is loaded into the EAX register (or AX or AL, if the size of the data is 16 bits or 8 bits, respectively).
CMOVZ x, y	If ZF = 1, the value stored in y is copied into x. If ZF = 0, the instruction acts like a no-op.
LEA x, y	The address y is stored in x. This is very much like the LC-3 instruction of the same name.

Table B.3 Control Instructions, x86 ISA

Instruction	Explanation
JMP x	IP is loaded with the address x. This is very much like the LC-3 instruction of the same name.
CALL x	The IP is pushed onto the stack, and a new IP is loaded with x.
RET	The stack is popped, and the value popped is loaded into IP.
LOOP x	ECX is decremented. If ECX is not 0 and ZF = 1, the IP is loaded with x.
INT n	The value n is an index into a table of descriptors that specify operating system service routines. The end result of this instruction is that IP is loaded with the starting result of the corresponding service routine. This is very much like the TRAP instruction in the LC-3.

Control The LC-3 has five control opcodes: BR, JSR/JSRR, JMP, RTI, and TRAP. x86 has all these and more. Table B.3 lists some of the control opcodes in the x86 instruction set.

Two Address versus Three Address

The LC-3 is a three-address ISA. This description reflects the number of operands explicitly specified by the ADD instruction. An add operation requires two source operands (the numbers to be added) and one destination operand, to store the result. In the LC-3, all three must be specified explicitly, hence the name three-address ISA.

Even if the same location is to be used both for one of the sources and for the destination, the three addresses are all specified. For example, the LC-3 ADD R1,R1,R2 identifies R1 as both a source and the destination.

The x86 is a two-address ISA. Since the add operation needs three operands, the location of one of the sources must also be used to store the result. For example, the corresponding ADD instruction in the x86 ISA would be ADD EAX, EBX. (EAX and EBX are names of two of the eight general purpose registers.) EAX and EBX are the sources, and EAX is the destination.

Since the result of the operate is stored in the location that originally contained one of the sources, that source operand is no longer available after that instruction is executed. If that source operand is needed later, it must be saved before the operate instruction is executed.

Memory Operands

A major difference between the LC-3 instruction set and the x86 instruction set is the restriction on where operate instructions can get their operands. An LC-3 operate instruction must obtain its source operands from registers and write the result to a destination register. An x86 instruction, on the other hand, can obtain one of its sources from memory and/or write its result to memory. In other words, the x86 can read a value from memory, operate on that value, and store the result in memory all in a single instruction. The LC-3 cannot.

The LC-3 program requires a separate load instruction to read the value from memory before operating on it, and a separate store instruction to write the result

in memory after the operate instruction. An ISA, like the LC-3, that has this restriction is called a *load-store* ISA. The x86 is not a load-store ISA.

B.1.2 Memory

The LC-3 memory consists of 2^{16} locations, each containing 16 bits of information. We say the LC-3 has a 16-bit address space, since one can uniquely address its 2^{16} locations with 16 bits of address. We say the LC-3 has an addressability of 16 bits, since each memory location contains 16 bits of information.

The x86 memory has a 32-bit address space and an addressability of eight bits. Since one byte contains eight bits, we say the x86 memory is byte addressable. Since each location contains only eight bits, four contiguous locations in memory are needed to store a 32-bit data element, say locations X, X+1, X+2, and X+3. We designate X as the address of the 32-bit data element. In actuality, X only contains bits [7:0], X+1 contains bits [15:8], X+2 contains bits [23:16], and X+3 contains bits [31:24] of the 32-bit value.

One can determine an LC-3 memory location by simply obtaining its address from the instruction, using one of the three addressing modes available in the instruction set. An x86 instruction has available to it more than two dozen addressing modes that it can use to specify the memory address of an operand. We examine the addressing modes in Section B.2 in the context of the x86 instruction format.

In addition to the larger number of addressing modes, the x86 contains a mechanism called *segmentation* that provides a measure of protection against unwanted accesses to particular memory addresses. The address produced by an instruction's addressing mode, rather than being an address in its own right, is used as an address within a segment of memory. Access to that memory location must take into account the segment register that controls access to that segment. The details of how the protection mechanism works will have to wait for later in your studies.

However, Figure B.2 does show how an address is calculated for the register+offset addressing mode, both for the LC-3, and for the x86, with segmentation. In both cases, the opcode is to move data from memory to a general purpose register. The LC-3 uses the LDR instruction. The x86 uses the MOV instruction. In the case of the x86, the address calculated is in the DS segment, which is accessed via the DS register. That access is done through a 16-bit *selector*, which indexes into a segment descriptor table, yielding the *segment descriptor* for that segment. The segment descriptor contains a *segment base register* and a *segment limit register*, and the protection information. The memory address obtained from the addressing mode of the instruction is added to the segment base register to provide the actual memory address, as shown in Figure B.2.

B.1.3 Internal State

The internal state of the LC-3 consists of eight 16-bit general purpose registers, R0 to R7, a 16-bit PC, and a 16-bit PSR that specifies the privilege mode, priority, and three 1-bit condition codes (N, Z, and P). The user-visible internal state of

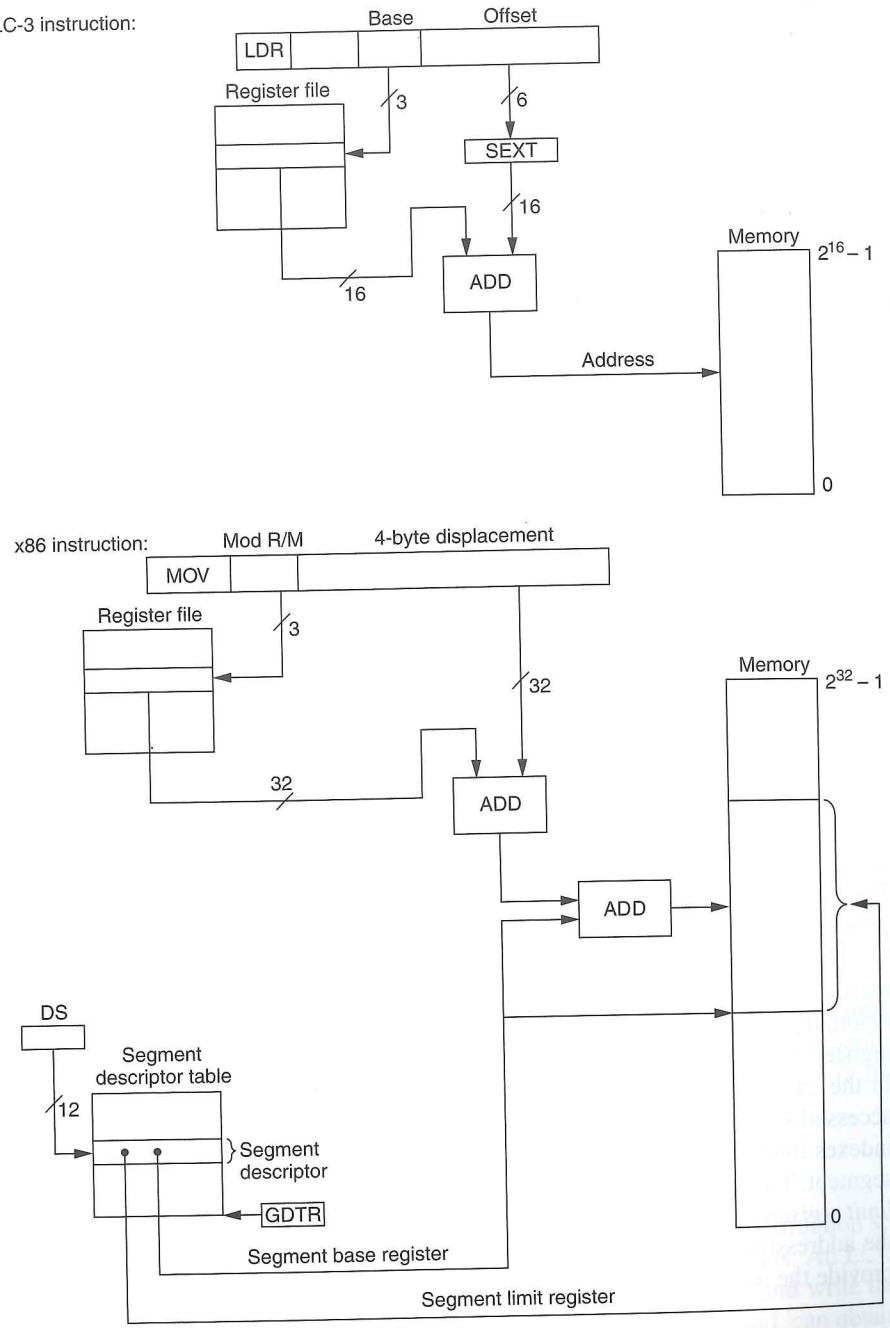

Figure B.2 Register+offset addressing mode in LC-3 and x86 ISAs

the x86 consists of application-visible registers, an Instruction pointer, a FLAGS register, and the segment registers.

Application-Visible Registers

Figure B.3 shows some of the application-visible registers in the x86 ISA.

Corresponding to R0 through R7, the x86 also has eight general purpose registers, EAX, EBX, ECX, EDX, ESP, EBP, ECI, and EDI. Each contains 32 bits, reflecting the normal size of its operands. However, since the x86 provides opcodes that process 16-bit operands and 8-bit operands, it should also provide 16-bit and 8-bit registers. The ISA identifies the low 16 bits of each 32-bit register as a 16-bit register and the low 8 bits and the high 8 bits of four of the registers as 8-bit registers for the use of instructions that require those smaller operands. So, for example, AX, BX, to DI are 16-bit registers, and AL, BL, CL, DL, AH, BH, CH, and DH are 8-bit registers.

The x86 also provides 64-bit registers for storing values needed for floating point and MMX computations. They are, respectively, FP0 through FP7 and MM0 through MM7.

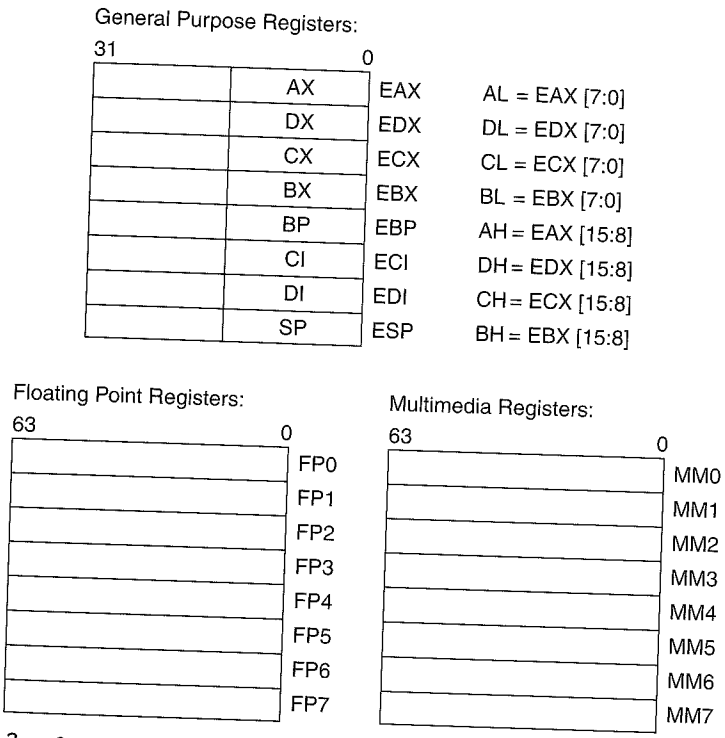

Figure B.3 Some x86 application-visible registers

System Registers

The LC-3 has two system-level registers—the PC and the PSR. The user-visible x86 has these and more.

Figure B.4 shows some of the user-visible system registers in the x86 ISA.

Instruction Pointer

The x86 has the equivalent of the LC-3's 16-bit program counter. The x86 calls it an *instruction pointer* (IP). Since the address space of the x86 is 32 bits, IP is a 32-bit register.

FLAGS Register

Corresponding to the LC-3's N, Z, and P condition codes, the x86 has a 1-bit SF (sign flag) register and a 1-bit ZF (zero flag) register. SF and ZF provide exactly the same functions as the N and Z condition codes of the LC-3. The x86 does not have the equivalent of the LC-3's P condition code. In fact, the P condition code is redundant, since if one knows the values of N and Z, one knows the value of P. We included it in the LC-3 ISA anyway, for the convenience of assembly language programmers and compiler writers.

The x86 collects other 1-bit values in addition to N and Z. These 1-bit values (called *flags*) are contained in a 16-bit register called FLAGS. Several of these flags are discussed in the following paragraphs.

The CF flag stores the *carry* produced by the last relevant operation that generated a carry. As we said earlier, together with the ADC instruction, CF facilitates the generation of procedures, which allows the software to deal with larger integers than the ISA supports.

The OF flag stores an *overflow* condition if the last relevant operate generated a value too large to store in the available number of bits. Recall the discussion of overflow in Section 2.5.3.

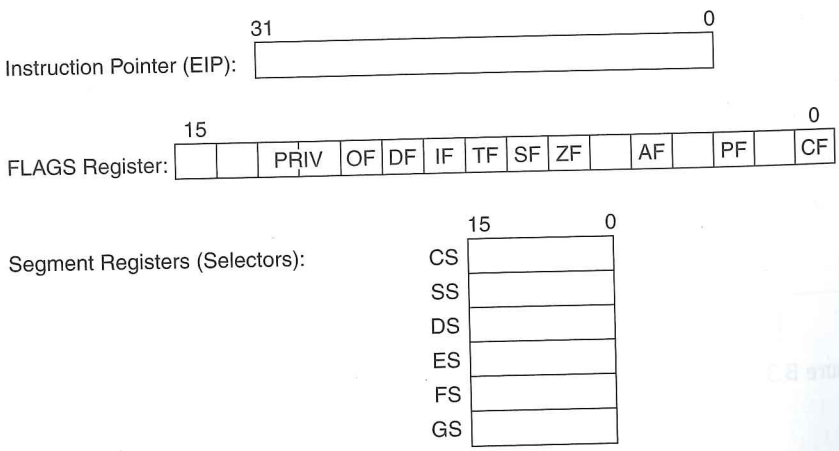

Figure B.4 x86 system registers

The DF flag indicates the *direction* in which string operations are to process strings. If DF = 0, the string is processed from the high-address byte down (i.e., the pointer keeping track of the element in the string to be processed next is decremented). If DF = 1, the string is processed from the low-address byte up (i.e., the string pointer is incremented).

Two flags not usually considered as part of the application state are the IF (*interrupt*) flag and the TF (*trap*) flag. Both correspond to functions with which you are familiar.

IF is very similar to the IE (interrupt enable) bit in the KBSR and DSR, discussed in Section 8.5. If IF = 1, the processor can recognize external interrupts (like keyboard input, for example). If IF = 0, these external interrupts have no effect on the process that is executing. We say the interrupts are *disabled*.

TF is very similar to *single-step mode* in the LC-3 simulator, only in this case it is part of the ISA. If TF = 1, the processor halts after every instruction so the state of the system can be examined. If TF = 0, the processor ignores the trap and processes the next instruction.

Segment Registers

When operating in its preferred operating mode (called *protected mode*), the address calculated by the instruction is really an offset from the starting address of a segment, which is specified by some *segment base register*. These segment base registers are part of their corresponding *data segment descriptors*, which are contained in the *segment descriptor table*. At each instant of time, six of these segments are active. They are called, respectively, the *code segment* (CS), *stack segment* (SS), and four data segments (DS, ES, FS, and GS). The six active segments are accessed via their corresponding segment registers shown in Figure B.4, which contain pointers to their respective segment descriptors.

B.2 The Format and Specification of x86 Instructions

The LC-3 instruction is a 16-bit instruction. Bits [15:12] always contain the opcode; the remaining 12 bits of each instruction are used to support the needs of that opcode.

The length of an x86 instruction is not fixed. It consists of a variable number of bytes, depending on the needs of that instruction. A lot of information can be packed into one x86 instruction. Figure B.5 shows the format of an

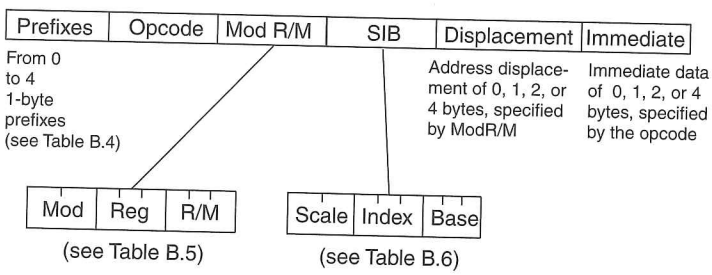

Figure B.5 Format of the x86 instruction

x86 instruction. The instruction consists of anywhere from 1 to 15 bytes, as shown in the figure.

The two key parts of an x86 instruction are the opcode and, where necessary, the ModR/M byte. The opcode specifies the operation the instruction is to perform. The ModR/M byte specifies how to obtain the operands it needs. The ModR/M byte specifies one of several addressing modes, some of which require the use of registers and a one-, two-, or four-byte displacement. The register information is encoded in a SIB byte. Both the SIB byte and the displacement (if one is necessary) follow the ModR/M byte in the instruction.

Some opcodes specify an immediate operand and also specify the number of bytes of the instruction that is used to store that immediate information. The immediate value (when one is specified) is the last element of the instruction.

Finally, the instruction assumes certain default information with respect to the semantics of an instruction, such as address size, operand size, segment to be used, and so forth. The instruction can change this default information by means of one or more prefixes, which are located at the beginning of the instruction.

Each part of an x86 instruction is discussed in more detail in Sections B.2.1 through B.2.6.

B.2.1 Prefix

Prefixes provide additional information that is used to process the instruction. There are four classes of prefix information, and each instruction can have from zero to four prefixes, depending on its needs. Fundamentally, a prefix overrides the usual interpretation of the instruction.

The four classes of prefixes are lock and repeat, segment override, operand override, and address override. Table B.4 describes the four types of prefixes.

Table B.4 Prefixes, x86 ISA

Repeat/Lock	
xF0 (LOCK)	This prefix guarantees that the instruction will have exclusive use of all shared memory until the instruction completes execution.
xF2, xF3 (REP/REPE/REPNE)	This prefix allows the instruction (a string instruction) to be repeated some specified number of times. The iteration count is specified by ECX. The instruction is also terminated on the occurrence of a specified value of ZF.
Segment override	
x2E(CS), x36(SS), x3E(DS), x26(ES), x64(FS), x65(GS)	This prefix causes the memory access to use the specified segment, instead of the default segment expected for that instruction.
Operand size override	
x66	This prefix changes the size of data expected for this instruction. That is, instructions expecting 32-bit data elements use 16-bit data elements. And instructions expecting 16-bit data elements use 32-bit data elements.
Address size override	
x67	This prefix changes the size of operand addresses expected for this instruction. That is, instructions expecting a 32-bit address use 16-bit addresses. And instructions expecting 16-bit addresses use 32-bit addresses.

B.2.2 Opcode

The opcode byte (or bytes—some opcodes are represented by two bytes) specifies a large amount of information about the needs of that instruction. The opcode byte (or bytes) specifies, among other things, the operation to be performed, whether the operands are to be obtained from memory or from registers, the size of the operands, whether or not one of the source operands is an immediate value in the instruction, and if so, the size of that immediate operand.

Some opcodes are formed by combining the opcode byte with bits [5:3] of the ModR/M byte, if those bits are not needed to provide addressing mode information. The ModR/M byte is described in Section B.2.3.

B.2.3 ModR/M Byte

The ModR/M byte, shown in Figure B.5, provides addressing mode information for two operands, when necessary, or for one operand, if that is all that is needed. If two operands are needed, one may be in memory, the other in a register, or both may be in registers. If one operand is needed, it can be either in a register or in memory. The ModR/M byte supports all cases.

The ModR/M byte is essentially partitioned into two parts. The first part consists of bits [7:6] and bits [2:0]. The second part consists of bits [5:3].

If bits [7:6] = 00, 01, or 10, the first part specifies the addressing mode of a memory operand, and the combined five bits ([7:6],[2:0]) identify which addressing mode. If bits [7:6] = 11, there is no memory operand, and bits [2:0] specify a register operand.

Bits [5:3] specify the register number of the other operand, if the opcode requires two operands. If the opcode only requires one operand, bits [5:3] are available as a subopcode to differentiate among eight opcodes that have the same opcode byte, as described in Section B.2.2.

Table B.5 lists some of the interpretations of the ModR/M byte.

Table B.5 ModR/M Byte, Examples

Mod	Reg	R/M	Eff. Addr.	Reg	Explanation
00	011	000	[EAX]	EBX	EAX contains the address of the memory operand. EBX contains the register operand.
01	010	000	disp8[EAX]	EDX	Memory operand's address is obtained by adding the displacement byte of the instruction to the contents of EAX. EDX contains the register operand.
10	000	100	disp32[-][-]	EAX	Memory operand's address is obtained by adding the four-byte (32 bits) displacement of the instruction to an address that will need an SIB byte to compute. (See Section B.2.4 for the discussion of the SIB byte.) EAX contains the register operand.
11	001	110	ESI	ECX	If the opcode requires two operands, both are in registers (ESI and ECX). If the opcode requires one operand, it is in ESI. In that case, 001 (bits [5:3]) are part of the opcode.

Table B.6 SIB Byte, Examples

Scale	Index	Base	Computation	Explanation
00	011	000	EBX+EAX	The contents of EBX are added to the contents of EAX. The result is added to whatever is specified by the ModR/M byte.
01	000	001	2 · EAX + ECX	The contents of EAX are multiplied by 2, and the result is added to the contents of ECX. This is then added to whatever is specified by the ModR/M byte.
01	100	001	ECX	The contents of ECX are added to whatever is specified by the ModR/M byte.
10	110	010	4 · ESI + EDX	The contents of ESI are multiplied by 4, and the result is added to the contents of EDX. This is then added to whatever is specified by the ModR/M byte.

B.2.4 SIB Byte

If the opcode specifies that an operand is to be obtained from memory, the ModR/M byte specifies the addressing mode, that is, the information that is needed to calculate the address of that operand. Some addressing modes require more information than can be specified by the ModR/M byte alone. Those operand specifiers (see example 3 in Table B.5) specify the inclusion of an SIB byte in the instruction. The SIB byte (for scaled-index-base), shown in Figure B.5, provides scaling information and identifies which register is to be used as an index register and/or which register is to be used as a base register. Taken together, the SIB byte computes scale · index + base, where base and/or index can be zero, and scale can be 1. Table B.6 lists some of the interpretations of the SIB byte.

B.2.5 Displacement

If the ModR/M byte specifies that the address calculation requires a displacement, the displacement (one, two, or four bytes) is contained in the instruction. The opcode and/or ModR/M byte specifies the size of the displacement.

Figure B.6 shows the addressing mode calculation for the source operand if the instruction is as shown. The prefix x26 overrides the segment register and specifies using the ES segment. The ModR/M and SIB bytes specify that a four-byte displacement is to be added to the base register ECX + the index register EBX after its contents are multiplied by 4.

B.2.6 Immediate

Recall that the LC-3 allowed small immediate values to be present in the instruction, by setting inst[5:5] to 1. The x86 also permits immediate values in the instruction. As stated previously, if the opcode specifies that a source operand is an immediate value in the instruction, it also specifies the number of bytes of the instruction used to represent the operand. That is, an immediate can be represented in the instruction with one, two, or four bytes. Since the opcode also specifies the size of the operand, immediate values that can be stored in fewer bytes than the

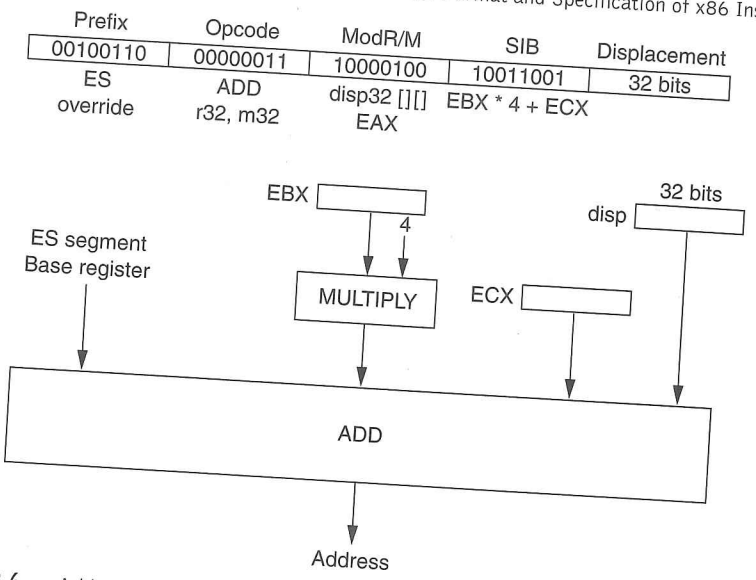

Figure B.6 Addressing mode calculation for Base+ScaledIndes+disp32

operand size are first sign-extended to their full size before being operated on. Figure B.7 shows the use of the immediate operand with the ADD instruction. The example is ADD EAX, $5. We are very familiar with the corresponding LC-3 instruction: ADD R0,R0,#5.

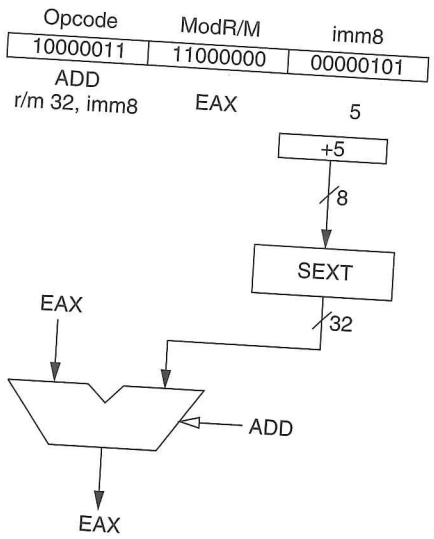

Figure B.7 Example x86 instruction: ADD EAX, $5

B.3 An Example

We conclude this appendix with an example. The problem is one we have dealt with extensively in Chapter 14. Given an input character string consisting of text, numbers, and punctuation, write a C program to convert all the lowercase letters to uppercase. Figure B.8 shows a C program that solves this problem. Figure B.9 shows the annotated LC-3 assembly language code that a C compiler would generate. Figure B.10 shows the corresponding annotated x86 assembly language code. For readability, we show assembly language representations of the LC-3 and x86 programs rather than the machine code.

```
#include <stdio.h>

void UpcaseString(char inputString[]);

main ()
{
    char string[8];

    scanf("%s", string);
    UpcaseString(string);
}

void UpcaseString(char inputString[])
{
  int i = 0;

  while(inputString[i]) {
    if (('a' <= inputString[i]) && (inputString[i] <= 'z'))
      inputString[i] = inputString[i] - ('a' - 'A');
    i++;
  }
}
```

Figure B.8 C source code for the upper-/lowercase program

```
; uppercase:  converts lower- to uppercase
            .ORIG  x3000
            LEA    R6, STACK
MAIN        ADD    R1, R6, #3
READCHAR    IN                      ; read in input string: scanf
            OUT
            STR    R0, R1, #0
            ADD    R1, R1, #1
            ADD    R2, R0, x-A
            BRnp   READCHAR
            ADD    R1, R1, #-1
            STR    R2, R1, #0       ; put in NULL char to mark the "end"
            ADD    R1, R6, #3       ; get the starting address of the string
            STR    R1, R6, #14      ; pass the parameter
            STR    R6, R6, #13
            ADD    R6, R6, #11
            JSR    UPPERCASE
            HALT
UPPERCASE   STR    R7, R6, #1
            AND    R1, R1, #0
            STR    R1, R6, #4
            LDR    R2, R6, #3
CONVERT     ADD    R3, R1, R2       ; add index to starting addr of string
            LDR    R4, R3, #0
            BRz    DONE             ; Done if NULL char reached
            LD     R5, a
            ADD    R5, R5, R4       ; 'a' <= input string
            BRn    NEXT
            LD     R5, z
            ADD    R5, R4, R5       ; input string <= 'z'
            BRp    NEXT
            LD     R5, asubA        ; convert to uppercase
            ADD    R4, R4, R5
            STR    R4, R3, #0
NEXT        ADD    R1, R1, #1       ; increment the array index, i
            STR    R1, R6, #4
            BRnzp  CONVERT
DONE        LDR    R7, R6, #1
            LDR    R6, R6, #2
            RET
a           .FILL  #-97
z           .FILL  #-122
asubA       .FILL  #-32
STACK       .BLKW  100
            .END
```

Figure B.9 LC-3 assembly language code for the upper-/lowercase program

```
.386P
.model FLAT

_DATA   SEGMENT                 ; The NULL-terminated scanf format
$SG397  DB      '%s', 00H       ; string is stored in global data space.
_DATA   ENDS

_TEXT   SEGMENT

_string$ = -8                   ; Location of "string" in local stack
_main   PROC NEAR
        sub   esp, 8            ; Allocate stack space to store "string"
        lea   eax, DWORD PTR _string$[esp+8]
        push  eax               ; Push arguments to scanf
        push  OFFSET FLAT:$SG397
        call  _scanf

        lea   ecx, DWORD PTR _string$[esp+16]
        push  ecx               ; Push argument to UpcaseString
        call  _UpcaseString

        add   esp, 20           ; Release local stack space
        ret   0
_main   ENDP

_inputString$ = 8               ; "inputString" location in local stack
_UpcaseString PROC NEAR
        mov   ecx, DWORD PTR _inputString$[esp-4]
        cmp   BYTE PTR [ecx], 0
        je    SHORT $L404           ; If inputString[0]==0, skip the loop
$L403:  mov   al, BYTE PTR [ecx]    ; Load inputString[i] into AL
        cmp   al, 97                ; 97 == 'a'
        jl    SHORT $L405
        cmp   al, 122               ; 122 == 'z'
        jg    SHORT $L405
        sub   al, 32                ; 32 == 'a' - 'A'
        mov   BYTE PTR [ecx], al
$L405:  inc   ecx                   ; i++ %$
        mov   al, BYTE PTR [ecx]
        test  al, al
        jne   SHORT $L403           ; Loop if inputString[i] != 0
$L404:  ret   0
_UpcaseString ENDP
_TEXT   ENDS
END
```

Figure B.10 x86 assembly language code for the upper-/lowercase program